МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»

ОТЧЕТ ПО УЧЕБНОЙ ПРАКТИКЕ

«Сортировка Шелла с чётно-нечётным слиянием Бэтчера»

Выполнил: студент группы 381706-1
Лембриков Степан Андреевич
_____ Подпись

Проверил:
Доцент кафедры МОСТ, кандидат технических наук
_____ Сысоев А. В.

Нижний Новгород
2019

Содержание

1. Введение

Для понимания данной работы в будущем необходимо рассказать о таком понятии, как сортировка Шелла.

Сортировка Шелла – алгоритм сортировки, идея которого состоит в сравнении элементов, стоящих на определённом расстоянии друг от друга. При данной сортировке сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии d, выбор которого зависит от версии алгоритма (в изначальном варианте сортировки Шелла оно равно n / 2, где n – число элементов в сортируемом массиве). После этого процедура сравнения повторяется для меньших значений d (опять же, первоначально размер d уменьшался вдвое на каждой итерации) и завершается тогда, когда d становится равным единице. Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы “быстрее” встают на свои места.

Лабораторная работа направлена на реализацию параллельного алгоритма сортировки, а именно - сортировки Шелла с чётно-нечётным слиянием Бэтчера.

Вообще говоря, можно выделить два варианта реализации распараллеливания сортировки: внутренняя реализация параллельного алгоритма и внешнее распараллеливание за счёт слияния отсортированных частей.

Данный алгоритм относится ко второму варианту, а, следовательно, он состоит из следующих шагов:

1. Сортировка частей массива.
2. Слияние отсортированных частей массива.

Более подробно о каждом из этих шагов, будет рассказано позднее в описании алгоритмов. Так же будет приведён пример сортировки Шелла.

2. Постановка задачи

В рамках данной лабораторной работы, как прежде уже было сказано, ставится задача эффективной реализации параллельного алгоритма сортировки Шелла с чётно-нечётным слиянием Бэтчера.

Для работы данного алгоритма необходимо реализовать следующие:

- Разбиение массива, который подаётся на вход, в зависимости от количества процессов, на части, далее именуемые мини-массивами
- Сортировка чётных элементов двух очередных мини-массивов
- Сортировка нечётных элементов двух очередных мини-массивов
- Слияние чётных и нечётных элементов двух мини-массивов в один
- Проход по полученному массиву для окончательной сортировки

3. Руководство программиста

3.1. Описание структуры программы

Программа состоит из следующих модулей:

- Модули lembrikov_s_shell_betch_mpi и lembrikov_s_shell_betch_mpi_lib. Включают в себя файлы shell_betch.h, в котором описаны методы, использующиеся в программе, и shell_betch.cpp, в котором написана их реализации. Так же есть файл main.cpp, который содержит набор тестов для проверки работоспособности написанной программы.

3.2 Описание алгоритмов

О самой сортировке Шелла, как об основе, которая необходима для данной лабораторной работы, уже было сказано в введении, поэтому здесь будет представлен лишь пример, который наглядно покажет работу данного алгоритма.

Пусть дан массив A=(32,95,16,82,24,66,35,19,75,54,40,43,93,68), и выполняется его сортировка методом Шелла, а в качестве значений d выбраны 5,3,1.

На первом шаге сортируются подмассивы A, составленные из всех элементов A, различающихся на 5 позиций, то есть подмассивы $A_{5,1}$ = (32, 66, 40), $A_{5,2}$ = (95, 35, 43), $A_{5,3}$ = (16, 19, 93), $A_{5,4}$ = (82, 75, 68), $A_{5,5}$ = (24, 54).

В полученном списке на втором шаге вновь сортируются подсписки из отстоящих на 3 позиции элементов.

Процесс завершается обычной сортировкой вставками получившегося списка.

Исходный массив	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
После сортировки с шагом 5	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 обменов
После сортировки с шагом 3	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 обменов
После сортировки с шагом 1	16	19	24	32	35	40	43	54	66	68	75	82	93	95	15 обменов

Рисунок 1. Сортировка Шелла

Теперь расскажем про чётно-нечетное слияние Бэтчера

Чётно-нечётное слияние Бэтчера заключается в том, что два упорядоченных массива, которые необходимо слить, разделяются на чётные и нечётные элементы. Такое слияние может быть выполнено параллельно. Чтобы массив стал окончательно отсортированным, достаточно сравнить пары элементов, стоящие на нечётной и чётной позициях.

Чётно-нечётное слияние Бэтчера позволяет задействовать 2 потока при слиянии двух упорядоченных массивов. В этом случае слияние n массивов могут выполнять n параллельных потоков. На следующем шаге слияние n/2 полученных массивов будут выполнять n/2 потоков и т.д. На последнем шаге два массива будут сливать 2 потока.

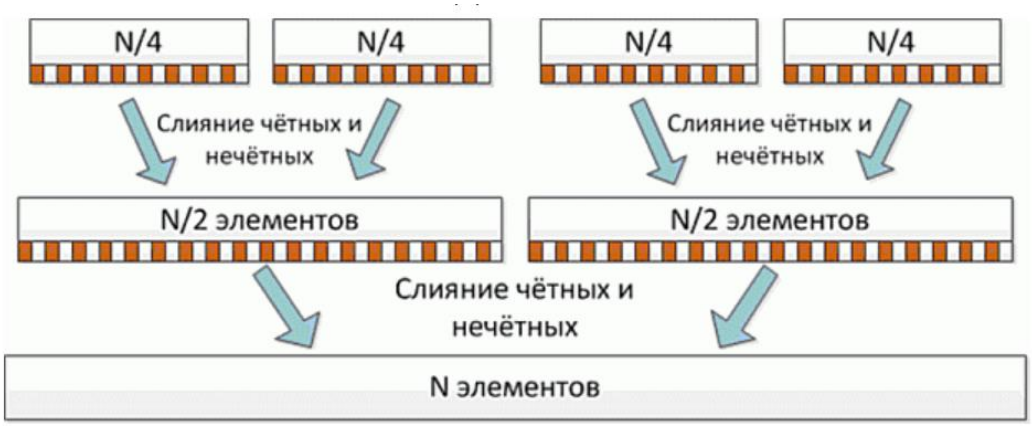

Рисунок 2. Слияние Бэтчера

4. Схема распараллеливания

Схема распараллеливания следующая:

- Сначала исходный массив делится на части по числу процессов. Эти части рассылаются по процессам и сортируется сортировкой Шелла. После этого процессы разбиваются на пары и пересылают друг другу свои отсортированные части. Если число процессов нечётное, то последний процесс отправляет свою отсортированную часть всем остальным.
- На этом шаге программа идёт в цикле по каждому биту числа процессов (из этих битов число можно разложить на степени двойки). На каждой итерации данного цикла сливаются очередные части массивов до тех пор, пока не закончатся биты, равные единице. Если число процессов равно степени двойки, то итерация будет одна, и мы получим полностью отсортированный массив. В противном случае, к примеру, при числе процессов, равном 7 (7 = 4 + 2 + 1) в конце этого шага мы получим три отсортированных массива, которые всё ещё надо слить.
- Данный шаг нужен в случае, описанном в конце предыдущего. Программа идёт ещё по одному циклу, сливая полученные массивы. (число итераций равно количеству бит в числе процессов, равных единице)

5. Описание программной реализации

Программа содержит 7 методов:

- getOtrVector – принимает число n и создаёт вектор такой же длины, заполненный чередующимися единицами и минус единицами. Используется в тестах.
- getRandomVector – принимает число n и создаёт вектор такой же длины, заполенный случайными числами от 0 до n.
- Chet_Betch – принимает два массива (каждый из которых должен быть отсортированным) и возвращает отсортированный массив, в котором находятся только элементы, стоящие на чётных позициях в исходных. Нужен для работы основного алгоритма.
- Nechet_Betch – принимает два массива (каждый из которых должен быть отсортированным) и возвращает отсортированный массив, в котором находятся только элементы, стоящие на нечётных позициях в исходных. Нужен для работы основного алгоритма.
- Sravnenie_Chet_Nechet – принимает два массива (каждый из которых должен быть отсортированным) и возвращает отсортированный массив, состоящий из элементов исходных массивов. Нужен для работы основного алгоритма.
- ShellSort – простой алгоритм сортировки Шелла. Используется для работы основного алгоритма, а также для работы тестов.
- Shell – основной алгоритм, использующийся для работы программы. Расписан в разделах “Описание алгоритмов” и “Схема распараллеливания”.

6. Подтверждение корректности

Для подтверждения корректности было написано 6 тестов.

В первых двух массивы заполнялись не с помощью какого-либо метода, а вручную (на малом количестве элементов). Результирующий массив для проверки также заполнялся вручную. Результат работы программы сравнивался именно с этим массивом. Разница между тестами лишь в том, что один сравнивал массивы на идентичность, другой на то, что массивы должны быть различны.

Ещё в двух тестах массивы заполнялись с помощью метода “getOtrVector”, последняя двойка – с помощью метода “getRandomVector ”. Массив для проверки заполнялся с помощью метода “Shell” (данные методы описаны в разделе “Описание программой реализации”).

Тесты были написаны с помощью библиотеки Google Testing Framework 3, их успешное выполнение и подтверждает работоспособность написанной программы.

7. Эксперименты

Эксперименты проводились на ПК со следующими параметрами:

- Операционная система: Windows 10 Домашняя
- Процессор: Intel(R) Core™ i5-8250U CPU @ 1.60 GHz
- ОЗУ 8 гб
- Версия Visual Studio: 2017

Эксперименты проводились на основе следующих данных:

Размер массива n = 30 млн. элементов. Количество процессов: 1 – 4.

Количество процессов	Время работы программы без использования параллельного алгоритма	Время работы программы с использованием параллельного алгоритма	Ускорение
1	9.95923	10.0942	1.013
2	10.6777	5.51673	0.516
3	10.1627	4.25181	0.418
4	10.4644	3.73282	0.356
5	10.3287	4.76244	0.461

Таблица 1. Время работы программы с и без использования параллельного алгоритма

Проведены эксперименты, в ходе которых было доказано, что алгоритм реализован верно и получена действительно эффективно работающая программа, ускорение которой практически сравнимо с линейным при количестве процессов, равным 2.

При дальнейшем увеличении количества процессов до 4, ускорение наблюдается, но оно не равно линейному. Это происходит из-за увеличения накладных расходов: приходится выделять больше памяти под отсортированные части массива, происходит больше пересылок данных между процессами.

При количестве процессов, больше 4, увеличение ускорения не наблюдается, даже наоборот, заметна небольшая деградация. Это связано как с накладными расходами, так и с ограниченностью параметров процессора, на котором проводились эксперименты.

8. Заключение

В ходе выполнения лабораторной работы я ещё раз вспомнил сортировку Шелла, которая изучалась ещё на первом курсе, узнал и разобрал алгоритм распараллеливания данной сортировки, а именно – алгоритм сортировки Шелла с чётно-нечётным слиянием Бэтчера.

Мною была написана программа, которая основывается на данном алгоритме и позволяет ускорить время работы обычной сортировки Шелла в два раза в лучшем случае.

Были написаны тесты, с помощью которых проводился контроль корректности.

9. Литература

- Ссылки в Internet
    1. Википедия. [https://ru.wikipedia.org/wiki/Сортировка Шелла]
    2. Интуит. [https://www.intuit.ru/studies/courses/Чётно-Нечётное слияние Бэтчера]

10. Приложение

1) Main.cpp:

```
// Copyright 2019 Lembrikov Stepan
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <vector>
#include "./shell_betch.h"

TEST(Mat_On_Vec_MPI, Test_On_Vector_EQ) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector <int> a(8);
    std::vector <int> res = {2, 3, 4, 5, 6, 7, 8, 9};
    std::vector <int> res_vector(8, 0);
    a[0] = 9;
    a[1] = 8;
    a[2] = 7;
    a[3] = 6;
    a[4] = 5;
    a[5] = 4;
    a[6] = 3;
    a[7] = 2;
    res_vector = Shell(a);
    if (rank == 0) {
        EXPECT_EQ(res_vector, res);
    }
}

TEST(Mat_On_Vec_MPI, Test_On_Vector_NE) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector <int> a(8);
    std::vector <int> res = {2, 3, 4, 5, 6, 7, 9, 8};
    std::vector <int> res_vector(8, 0);
    a[0] = 9;
    a[1] = 8;
    a[2] = 7;
    a[3] = 6;
    a[4] = 5;
    a[5] = 4;
    a[6] = 3;
    a[7] = 2;
    res_vector = Shell(a);
    if (rank == 0) {
        EXPECT_NE(res_vector, res);
    }
}

TEST(Mat_On_Vec_MPI, Test_On_Otr_Vector_EQ) {
    int razmer = 16;
    std::vector <int> a = getOtrVector(razmer);
    std::vector <int> res(razmer);
    std::vector <int> res_vector(razmer);
    res = a;
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    res_vector = Shell(a);
    if (rank == 0) {
        res = ShellSort(res, razmer);
```

```
        EXPECT_EQ(res_vector, res);
    }
}

TEST(Mat_On_Vec_MPI, Test_On_Otr_Vector_NE) {
    int razmer = 16;
    std::vector <int> a = getOtrVector(razmer);
    std::vector <int> res(razmer);
    std::vector <int> res_vector(razmer);
    res = a;
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    res_vector = Shell(a);
    if (rank == 0) {
        res = ShellSort(res, razmer);
        res[razmer - 1] = -1;
        EXPECT_NE(res_vector, res);
    }
}

TEST(Mat_On_Vec_MPI, Test_On_Random_Vector_EQ) {
    int razmer = 30000000;
    std::vector <int> a = getRandomVector(razmer);
    std::vector <int> res_vector_multy(razmer, 0);
    std::vector <int> res(razmer, 0);
    res = a;
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     double start_multy = MPI_Wtime();
    res_vector_multy = Shell(a);
     double end_multy = MPI_Wtime();

    if (rank == 0) {
         double start_solo = MPI_Wtime();
        res = ShellSort(res, razmer);
         double end_solo = MPI_Wtime();
        EXPECT_EQ(res_vector_multy, res);
         std::cout << end_solo - start_solo << "\n";
         std::cout << end_multy - start_multy << "\n";
    }
}

TEST(Mat_On_Vec_MPI, Test_On_Random_Vector_NE) {
    int razmer = 36;
    std::vector <int> a = getRandomVector(razmer);
    std::vector <int> res_vector_multy(razmer, 0);
    std::vector <int> res(razmer, 0);
    res = a;
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    res_vector_multy = Shell(a);

    if (rank == 0) {
        res = ShellSort(res, razmer);
        res[razmer - 1] = -1;
        EXPECT_NE(res_vector_multy, res);
    }
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    MPI_Init(&argc, &argv);
```

```
    ::testing::AddGlobalTestEnvironment(new GTestMPIListener::MPIEnvironment);
    ::testing::TestEventListeners& listeners =
        ::testing::UnitTest::GetInstance()->listeners();

    listeners.Release(listeners.default_result_printer());
    listeners.Release(listeners.default_xml_generator());

    listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);

    return RUN_ALL_TESTS();
}
```

2) Shell_betch.h:

```
// Copyright 2019 Lembrikov Stepan
#ifndef MODULES_TASK_3_LEMBRIKOV_S_SHELL_BETCH_SHELL_BETCH_H_
#define MODULES_TASK_3_LEMBRIKOV_S_SHELL_BETCH_SHELL_BETCH_H_

#include <mpi.h>
#include <iostream>
#include <random>
#include <vector>

std::vector<int> getRandomVector(int n);
std::vector<int> getOtrVector(int n);
//std::vector<int> getPoslVector(int n);
std::vector<int> ShellSort(const std::vector <int> &mas, int size);
std::vector<int> Shell(std::vector <int> mas);

#endif  // MODULES_TASK_3_LEMBRIKOV_S_SHELL_BETCH_SHELL_BETCH_H_
```

3) Shell_betch.cpp:

```
// Copyright 2019 Lembrikov Stepan
#include <../../../modules/task_3/lembrikov_s_shell_betch/shell_betch.h>
#include <algorithm>
#include <vector>
#include <iomanip>

std::vector<int> getOtrVector(int n) {
    std::vector <int> a(n);
    int k = 1;
    for (int i = 0; i < n; i++) {
        k = -k;
        a[i] = 1 * k;
    }
    return a;
}

std::vector<int> getRandomVector(int n) {
    // std::mt19937 engine;
    std::vector<int> a(n);
    std::mt19937 gen(time(0));
    // std::uniform_real_distribution<> urd(0, 1);
    std::uniform_int_distribution <int> urd(0, n);
    // engine.seed(n);
    // int k = 1;
    for (int i = 0; i < n; i++) {
        // k = -k;
        a[i] = urd(gen);
    }
```

```
    return a;
}

std::vector <int> Chet_Betch(const std::vector <int> &mas_1,
    const std::vector <int> &mas_2) {
    int size1 = mas_1.size();
    int size2 = mas_2.size();
    int size_res = size1 / 2 + size2 / 2 + size1 % 2 + size2 % 2;
    std::vector <int> mas_res(size_res);
    int it1 = 0;
    int it2 = 0;
    int i = 0;

    while ((it1 < size1) && (it2 < size2)) {
        if (mas_1[it1] <= mas_2[it2]) {
            mas_res[i] = mas_1[it1];
            it1 += 2;
        } else {
            mas_res[i] = mas_2[it2];
            it2 += 2;
        }
        i++;
    }

    if (it1 >= size1) {
        for (int j = it2; j < size2; j += 2) {
            mas_res[i] = mas_2[j];
            i++;
        }
    } else {
        for (int j = it1; j < size1; j += 2) {
            mas_res[i] = mas_1[j];
            i++;
        }
    }
    return mas_res;
}

std::vector <int> Nechet_Betch(const std::vector <int> &mas_1,
    const std::vector <int> &mas_2) {
    int size1 = mas_1.size();
    int size2 = mas_2.size();
    int size_res = size1 / 2 + size2 / 2;
    std::vector <int> mas_res(size_res);
    int it1 = 1;
    int it2 = 1;
    int i = 0;

    while ((it1 < size1) && (it2 < size2)) {
        if (mas_1[it1] <= mas_2[it2]) {
            mas_res[i] = mas_1[it1];
            it1 += 2;
        } else {
            mas_res[i] = mas_2[it2];
            it2 += 2;
        }
        i++;
    }

    if (it1 >= size1) {
        for (int j = it2; j < size2; j += 2) {
            mas_res[i] = mas_2[j];
```

```
            i++;
        }
    } else {
        for (int j = it1; j < size1; j += 2) {
            mas_res[i] = mas_1[j];
            i++;
        }
    }
    return mas_res;
}

std::vector <int> Sravnenie_Chet_Nechet(const std::vector <int> &mas_res_1,
    const std::vector <int> &mas_res_2) {
    int size1 = mas_res_1.size();
    int size2 = mas_res_2.size();
    int flag = 0;
    if (size1 - size2 == 2)
        flag = 1;
    if (size2 - size1 == 2)
        flag = 2;
    int size_res = mas_res_1.size() + mas_res_2.size();
    int size_min = size_res / 2 - 1;
    std::vector <int> mas_res(size_res);
    int buf;
    int i = 0;
    if (flag == 0) {
        for (i = 0; i < size1; i++) {
            mas_res[2 * i] = mas_res_1[i];
            mas_res[2 * i + 1] = mas_res_2[i];
        }
    }
    if ((flag == 1) || (flag == 2)) {
        for (i = 0; i < size_min; i++) {
            mas_res[2 * i] = mas_res_1[i];
            mas_res[2 * i + 1] = mas_res_2[i];
        }
    }

    if (flag == 1) {
        mas_res[2 * i] = mas_res_1[i];
        mas_res[2 * i + 1] = mas_res_1[i + 1];
    }

    if (flag == 2) {
        mas_res[2 * i] = mas_res_2[i];
        mas_res[2 * i + 1] = mas_res_2[i + 1];
    }

    for (int i = 1; i < size_res; i++) {
        if (mas_res[i] < mas_res[i - 1]) {
            buf = mas_res[i - 1];
            mas_res[i - 1] = mas_res[i];
            mas_res[i] = buf;
        }
    }

    return mas_res;
}

std::vector <int> ShellSort(const std::vector <int> &mas, int size) {
    int step, i, j, tmp;
    std::vector <int> mas_res(mas);
    for (step = size / 2; step > 0; step /= 2)
```

```
        for (i = step; i < size; i++)
            for (j = i - step; j >= 0 && mas_res[j] > mas_res[j + step]; j -= step) {
                tmp = mas_res[j];
                mas_res[j] = mas_res[j + step];
                mas_res[j + step] = tmp;
            }
    return mas_res;
}

std::vector <int> Shell(std::vector <int> mas) {
    int size;
    int rank;
    int ost;
    int flag = 0;
    int ostatok = 0;
    int k = 0;
    int size_mas = mas.size();
    if (size_mas == 1)
        return mas;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if ((size % 2 == 1) && (size > 1)) {
        flag = -1;
    }
    else if ((size > size_mas) || (size == 1)) {
        flag = 1;
    }
    ostatok = size_mas % size;
    k = size_mas / size;
    std::vector <int> part_mas(k + ostatok, 0);
    MPI_Status status;

    MPI_Bcast(&mas[0], size_mas, MPI_INT, 0, MPI_COMM_WORLD);


    if (flag == 1) {
        mas = ShellSort(mas, size_mas);
        return mas;
    }

    if (flag == 0) {
        for (int i = 0; i < size - 2; i += 2) {
            if (rank == i) {
                part_mas = ShellSort({ mas.cbegin() + i * k, mas.cbegin() + (i + 1) * k
}, k);
                std::copy(part_mas.begin(), part_mas.begin() + k, mas.begin() + i * k);
                if (i < size - 1) {
                    MPI_Sendrecv(&part_mas[0], k, MPI_INT, i + 1, 0,
                        &mas[(i + 1) * k], k, MPI_INT, i + 1, 0, MPI_COMM_WORLD,
&status);
                }
            }
        }
        for (int i = 1; i < size - 1; i += 2) {
            if (rank == i) {
                part_mas = ShellSort({ mas.cbegin() + i * k, mas.cbegin() + (i + 1) * k
}, k);
                std::copy(part_mas.begin(), part_mas.begin() + k, mas.begin() + k * i);
                MPI_Sendrecv(&part_mas[0], k, MPI_INT, i - 1, 0,
                    &mas[(i - 1) * k], k, MPI_INT, i - 1, 0, MPI_COMM_WORLD, &status);
            }
        }
```

```
        if (rank == size - 2) {
            part_mas = ShellSort({ mas.cend() - 2 * k - ostatok, mas.cend() - k - ostatok
}, k);
            std::copy(part_mas.begin(), part_mas.begin() + k, mas.end() - 2 * k -
ostatok);
            MPI_Sendrecv(&part_mas[0], k, MPI_INT, size - 1, 0,
                &mas[(size - 1) * k], k + ostatok, MPI_INT, size - 1, 0, MPI_COMM_WORLD,
&status);
        }

        if (rank == size - 1) {
            part_mas = ShellSort({ mas.cend() - k - ostatok, mas.cend() }, k + ostatok);
            std::copy(part_mas.begin(), part_mas.begin() + k + ostatok, mas.end() - k -
ostatok);
            MPI_Sendrecv(&part_mas[0], k + ostatok, MPI_INT, size - 2, 0,
                &mas[(size - 2) * k], k, MPI_INT, size - 2, 0, MPI_COMM_WORLD, &status);
        }
    } else if (flag == -1) {
        for (int i = 0; i < size - 1; i += 2) {
            if (rank == i) {
                part_mas = ShellSort({ mas.cbegin() + i * k, mas.cbegin() + (i + 1) * k
}, k);
                std::copy(part_mas.begin(), part_mas.begin() + k, mas.begin() + i * k);
                if (i < size - 1) {
                    MPI_Sendrecv(&part_mas[0], k, MPI_INT, i + 1, 0,
                        &mas[(i + 1) * k], k, MPI_INT, i + 1, 0, MPI_COMM_WORLD,
&status);
                }
            }
        }
        for (int i = 1; i < size; i += 2) {
            if (rank == i) {
                part_mas = ShellSort({ mas.cbegin() + i * k, mas.cbegin() + (i + 1) * k
}, k);
                std::copy(part_mas.begin(), part_mas.begin() + k, mas.begin() + k * i);
                MPI_Sendrecv(&part_mas[0], k, MPI_INT, i - 1, 0,
                    &mas[(i - 1) * k], k, MPI_INT, i - 1, 0, MPI_COMM_WORLD, &status);
            }
        }

        if (rank == size - 1) {
            part_mas = ShellSort({ mas.cend() - k - ostatok, mas.cend() }, k + ostatok);
            std::copy(part_mas.begin(), part_mas.begin() + k + ostatok, mas.end() - k -
ostatok);
        }
        MPI_Bcast(&mas[k * (size - 1)], k + ostatok, MPI_INT, size - 1, MPI_COMM_WORLD);
    }

    int iter = 0;
    int count_iter = 0;

    int kolvo_bit = sizeof(int) * 8;
    std::vector <int> bit(kolvo_bit);
    int it_bit = 0;
    for (int i = kolvo_bit - 1; i >= 0; i--) {
        if ((static_cast<bool>((1 << i)  &  size)) == 1) {
            bit[it_bit] = pow(2, i);
            it_bit++;
        }
    }
    int flag_prim_ost = 0;
    int count_otsort_mas = 0;
    if (size > 1) {
```

```
        int p = 0;
        while (p != it_bit) {
            if (bit[p] == 1) {
                break;
            }
            k = size_mas / size;
            count_iter = 0;
            count_otsort_mas += bit[p];
            if (count_otsort_mas == size) {
                flag_prim_ost = 1;
            }
            int buf = bit[p];
            while (buf != 1) {
                buf = buf / 2;
                count_iter++;
            }
            iter = 0;

            while (iter < count_iter) {
                if ((k % 2) == 1)
                    ost = 1;
                else
                    ost = 0;
                std::vector <int> res_part_mas1(k + ost);
                std::vector <int> res_part_mas2(k - ost);
                std::vector <int> res_part_mas(2 * k);
                int it_par = (count_otsort_mas - bit[p]) / 2;
                int smesh = (count_otsort_mas - bit[p]) * (size_mas / size);
                int dobavka = 0;
                int it = bit[p] / pow(2, (iter + 1));
                for (int i = 0; i < it; i++) {
                    if (i > 0) {
                        smesh += 2 * k;
                    }
                    if (i == (it - 1)) {
                        if (flag_prim_ost == 1) {
                            if (ost == 0) {
                                res_part_mas1.resize(k + ostatok / 2 + ostatok % 2);
                                res_part_mas2.resize(k + ostatok / 2);
                            }
                            if (ost == 1) {
                                res_part_mas1.resize(k + ost + ostatok / 2);
                                res_part_mas2.resize(k - ost + ostatok / 2 + ostatok %
2);

                            }
                            res_part_mas.resize(2 * k + ostatok);
                            dobavka = ostatok;
                        }
                    }

                    if (rank == it_par * 2) {
                        res_part_mas1 = Chet_Betch({ mas.cbegin() + smesh,
            mas.cbegin() + smesh + k },
                            { mas.cbegin() + smesh + k, mas.cbegin() + smesh + 2 * k +
dobavka });
                        MPI_Send(&res_part_mas1[0], res_part_mas1.size(), MPI_INT, it_par
* 2 + 1, 0, MPI_COMM_WORLD);
                    }
                    if (rank == it_par * 2 + 1) {
                        MPI_Status status0;
                        res_part_mas2 = Nechet_Betch({ mas.cbegin() + smesh,
            mas.cbegin() + smesh + k },
```

```
                            { mas.cbegin() + smesh + k, mas.cbegin() + smesh + 2 * k +
dobavka });
                        MPI_Recv(&res_part_mas1[0], res_part_mas1.size(), MPI_INT,
                            it_par * 2, 0, MPI_COMM_WORLD, &status0);
                        res_part_mas = Sravnenie_Chet_Nechet(res_part_mas1,
res_part_mas2);
                        MPI_Send(&res_part_mas[0], res_part_mas.size(), MPI_INT, 0, 1,
MPI_COMM_WORLD);
                    }
                    if (rank == 0) {
                        MPI_Status status1;
                        if ((size > 1) && ((it_par * 2 + 1) < size)) {
                            MPI_Recv(&mas[smesh], res_part_mas.size(),
                                MPI_INT, it_par * 2 + 1, 1, MPI_COMM_WORLD, &status1);
                        }
                    }
                    MPI_Bcast(&mas[smesh], res_part_mas.size(), MPI_INT, 0,
MPI_COMM_WORLD);
                    it_par++;
                }
                k = k * 2;
                iter++;
            }
            p++;
        }
    }
    if ((rank == 0) && (size > 2)) {
        int l = 0;
        std::vector<int> bit_sum(it_bit, 0);
        bit_sum[0] = bit[0];
        for (int i = 1; i < it_bit; i++) {
            bit_sum[i] = bit[i] + bit_sum[i - 1];
        }
        k = size_mas / size;
        while (l < (it_bit - 2)) {
            int k123 = ((bit[l + 1] + bit_sum[l]) / 2) * (size_mas / size);
            std::vector <int> res_part_mas1(k123);
            std::vector <int> res_part_mas2(k123);
            std::vector <int> res_part_mas(2 * k123);
            res_part_mas1 = Chet_Betch({ mas.cbegin(), mas.cbegin() + k * bit_sum[l] },
                { mas.cbegin() + k * bit_sum[l], mas.cbegin() + k * (bit_sum[l] + bit[l +
1]) });
            res_part_mas2 = Nechet_Betch({ mas.cbegin(), mas.cbegin() + k * bit_sum[l] },
                { mas.cbegin() + k * bit_sum[l], mas.cbegin() + k * (bit_sum[l] + bit[l +
1]) });
            res_part_mas = Sravnenie_Chet_Nechet(res_part_mas1, res_part_mas2);
            for (int i = 0; i < 2 * k123; i++) {
                mas[i] = res_part_mas[i];
            }
            l++;
        }
        if (it_bit > 1) {
            std::vector <int> res_part_mas1(size_mas / 2 + size_mas % 2);
            std::vector <int> res_part_mas2(size_mas / 2);
            res_part_mas1 = Chet_Betch({ mas.cbegin(), mas.cbegin() + k * bit_sum[l] },
                { mas.cbegin() + k * bit_sum[l], mas.cend() });
            res_part_mas2 = Nechet_Betch({ mas.cbegin(), mas.cbegin() + k * bit_sum[l] },
                { mas.cbegin() + k * bit_sum[l], mas.cend() });
            mas = Sravnenie_Chet_Nechet(res_part_mas1, res_part_mas2);
        }
    }
    return mas;
}
```